On Optimally Combining Static and Dynamic Analyses For Intensional Program Properties

Ravi Mangal     David Devecsergy     Alessandro Orso
Motivation

Recipe for designing automated, efficient, sound, and complete program verification tools
What is program verification?

Program $P$  
Spec $\phi$  
Verifier  

$[P] = [\phi]$  
$[P] \neq [\phi]$  
?
Static Verifiers - Type Systems

Program $P$

Type $\phi$

Type Inference/Checking

$\vdash P: \phi$

$\nu P: \phi$

$?$
Static Verifiers - Program Logics

Interactive Proof Checker

Program P

Spec φ

⊦ {T} P {φ}

'? {T} P {φ}

'?
Static Verifiers - Program Analyses

Diagram:
- Program P
- Spec φ
- Abstract Interpreter
- \([P] = [φ]\)
- ?
Dynamic Verifiers

Program P

Spec $\phi$

Instrumenter

Program P with assertions/contracts

Evaluator

Program Input

Fail

Pass
Let’s combine!

- Program P
- Spec $\phi_i$

Abstract Interpreter → ? → Instrumenter → P with assertions $\phi_i$

Evaluator
- Fail
- Pass

Program Input
Are we done?

Yes, and no :

- Yes, for ‘extensional’ or ‘local’ properties
  - Properties about the results of computations
  - Can be answered by observing the current program state

- No, for ‘intensional’ properties
  - Properties about how computations execute
Intensional Properties

- Track how computations execute

- Examples - taint tracking, datarace checking, program complexity, etc.

- Instrumented semantics - Augment the states of standard semantics with instrumentation data
  - Ex. $(P, S) \leadsto (P', S')$ modified to $(P, S, I) \leadsto (P', S', I')$
What is the problem?

- Combined verification only removes runtime assertions but not additional instrumentation
  - High overheads!

- No existing general framework for combining static and dynamic verifiers with a focus on efficiency
Idea!

- Remove “all” dynamic instrumentation “associated” with statically proven assertions

- Challenges:
  - How to communicate information from static to dynamic verifier?
  - How to formally define the notions of “all” and “associated” in the statement above?
  - How to automatically discover “all associated” instrumentation for a given assertion?

- Approach:
  - “Parameterize” static and dynamic verifiers
  - Notions of “all” and “associated” defined with respect to this parameterization
A taint analysis example

```c
void main() {
    A o1 = src1();
    B o2 = src2();
    A o3 = foo(o1);
    B o4 = bar(o2);
    assert(o3 not tainted);
    sink1(o3);
    assert(o4 not tainted);
    sink2(o4);
}
```
An interlude: Some definitions

eval : State \rightarrow State \quad \text{(evaluator for the language)}

abs_int : Abs_State \rightarrow Abs_State \quad \text{(abstract interpreter)}

\alpha : 2^{\text{State}} \rightarrow Abs_State \quad \text{(abstraction function)}

\alpha\{\text{eval}(s) | s \in X\} \subseteq \text{abs_int}(\alpha(X)) \quad \text{(relationship between eval and abs_int)}

erase : State \rightarrow \text{Uninstrumented_State}

extract : State \rightarrow \text{Instrumentation_Data}
Parametric Verifiers - Dynamic

\[ \text{eval}_\pi : \text{Parameter} \times \text{State} \rightarrow \text{State} \quad \text{(parametric evaluator for the language)} \]

\begin{align*}
\forall s,\pi. & \quad (\text{erase}(\text{eval}_\pi(\pi,s)) = \text{erase}(\text{eval}(s))) \quad \lor \quad (\text{eval}_\pi(\pi,s) = \text{fail}_i) \\
\forall \pi_1,\pi_2. & \quad \pi_1 \preceq \pi_2 \implies \exists s,i,j. (\text{eval}_\pi(\pi_1,s) = \text{fail}_i) \implies (\text{eval}_\pi(\pi_2,s) = \text{fail}_j) 
\end{align*}
Parametric Verifiers - Static

abs_int_\pi: \text{Parameter} \times \text{Abs\_State} \rightarrow \text{Abs\_State} \quad \text{(parametric abstract interpreter)}

\forall X, \pi. \quad \alpha\{\text{eval}_\pi(\pi,s) \mid s \in X\} \subseteq \text{abs_int}_\pi(\pi, \alpha(X))

\forall \pi_1, \pi_2. \quad \pi_1 \preceq \pi_2 \quad \Rightarrow \quad \text{fails}(\text{abs_int}_\pi(\pi_1, \text{init})) \subseteq \text{fails}(\text{abs_int}_\pi(\pi_2, \text{init}))
Let’s combine, again!

Program P

Abstract Interpreter

Instrumenter

P with assertions

Evaluator

Fail
Pass

Program Input

Parameter $\pi$

Assertions $\phi_i$
Minimal Parameter Problem (MPP)

- Find minimal parameter $\pi$ such that,
  \[ \forall s, i. \ (\text{eval}_\pi(T, s) = \text{fail}_i) \Rightarrow (\text{eval}_\pi(\pi, s) = \text{fail}_i) \]

- Finding the minimal parameter is equivalent to removing “all dynamic” instrumentation “associated” with statically proven assertions!

- MPP is not tractable, but we can approximately solve it by exploiting the relationship between our parametric static and dynamic verifiers
A simple parameter search algorithm
Analysis designer’s responsibilities

● Parameterize static and dynamic verifiers with a shared parametric notion

● Ensure:
  ○ $\alpha\{\text{eval}(s) | s \in X\} \subseteq \text{abs\_int}(\alpha(X))$
  ○ $\forall s, \pi. \quad (\text{erase}(\text{eval}_\pi(\pi, s)) = \text{erase}(\text{eval}(s))) \lor (\text{eval}_\pi(\pi, s) = \text{fail}_i)$
  ○ $\forall \pi_1, \pi_2. \quad \pi_1 \preceq \pi_2 \Rightarrow \exists s, i, j. (\text{eval}_\pi(\pi_1, s) = \text{fail}_i) \Rightarrow (\text{eval}_\pi(\pi_2, s) = \text{fail}_j)$
  ○ $\forall X, \pi. \quad \alpha\{\text{eval}_\pi(\pi, s) | s \in X\} \subseteq \text{abs\_int}_\pi(\pi, \alpha(X))$
  ○ $\forall \pi_1, \pi_2. \quad \pi_1 \preceq \pi_2 \Rightarrow \text{fails}(\text{abs\_int}_\pi(\pi_1, \text{init})) \subseteq \text{fails}(\text{abs\_int}_\pi(\pi_2, \text{init}))$

● And then,
Can we improve?

- **Improve parameter search:**
  - Exploit the structure of static verifier proofs
  - Leverage existing work on finding best abstractions

- **Further reduce dynamic verifier overheads**
  - Make optimistic/speculative assumptions statically to prove more assertions
  - Check these assumptions at runtime
  - Reduced instrumentation due to more proven assertions vs Overhead of checking assumptions

- **Connections to gradual typing and hybrid typing** (Please help!)
Thank You!